ERDC MSRC Major Shared Resource Center

ERDC MSRC/PET TR/01-01

Emulating Co-Array Fortran with Pthreads

by

Richard J. Hanson
Stephen F. Wornom

18 February 2001

01h00901

Work funded by the DoD High Performance Computing Modernization Program ERDC Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
CSC Nichols

Emulating Co-Array Fortran with Pthreads

Authors
Richard J. Hanson
Rice Center for High Performance Software Research
Rice University
6100 South Main Street
MS-41
Houston, TX 77005
Tel: 713-348-5304
e-mail: koolhans@rice.edu

Stephen F. Wornom
U.S. Army Engineer Research and Development Center
ATTN: Dr. Stephen Wornom CEERD-IH-C
3909 Halls Ferry Road
Vicksburg, MS 39180-6199
Tel: 601-634-4663
E-mail: wornomsf@wes.hpc.mil
Address all correspondence to this author

Emulating Co-Array Fortran with Pthreads

Richard J. Hanson, Rice University, Houston, TX
Stephen F. Wornom, U.S. Army Engineer Research and Development Center, Vicksburg, MS

Abstract

This report shows timing charts for a domain decomposition version of the Jacobi iterative method applied to solving a particular Helmholtz differential equation. Threads are used to emulate the algorithm specifications, which are given in Co-Array Fortran. A Fortran 90 API to Pthreads, developed in the U.S. Army Engineer Research and Development Center (ERDC) Major Shared Resource Center (MSRC) Programming Environment and Training program (PET), was used to make the tests. Typical results are shown for the SGI Origin 2000, SGI Origin 3000, SUN E10000, COMPAQ ES-40, and IBM Power3 super computer systems. The conversion from Co-Array Fortran to Fortran Pthreads is relatively easy. The timings show that thread programming can provide a portable and efficient alternative to Co-Array Fortran.

Introduction

Co-Array Fortran is proposed as an extension to Fortran 95 for expressing one-way communication between cooperating *program images*. This programming methodology is effective in expressing domain decomposition data exchange steps. In the example provided by Dr. Alan Wallcraft of the Naval Research Laboratory at the Stennis Space Center, a Jacobi iterative method, together with domain decomposition, is used to solve a linear system of algebraic equations that derives from a Helmholtz PDE problem, with periodic boundary conditions:

$$l^2u - \Delta u = f(x,y),\ (x,y) \in \text{ unit square}$$

Approximating this equation, using central second divided differences in both variables, and constant difference quotient h and $l^2 = 2h^{-2}$, yields the five-point stencil about each grid value,

$$\boxed{\begin{array}{ccc} 0 & -1 & 0 \\ -1 & 6 & -1 \\ 0 & -1 & 0 \end{array}}$$

(The constant at the midpoint of the stencil is arbitrary, except that it will exceed the value 4.) The choice made for l achieves a diagonally dominant example so that the Jacobi iteration converges in a few steps. This is reasonable because the interests are in measuring speedups of domain decomposition using the Fortran Pthreads API.

Normally, one would not use the Jacobi iteration on this problem because there are superior numerical algorithms. Finally, the function $f(x_i, y_j) = h^{-2}$ has the value one in the unit square and has the value zero elsewhere. This choice approximates a generalized or Dirac "delta" function at the distinguished point, (x_i, y_j).

Co-Array Operations Become Thread Operations

The unit square domain is decomposed into a $p \times q$ grid of nodes. This yields pq images that compute a component part of the Jacobi iteration. Each node carries a border or halo of values that are acquired from its neighbors. The need to communicate the halo values between blocks creates the problem of communication of the nodes with each other.

Using Co-Array Fortran, this communication of halo values in each Jacobi iteration is expressed with an extended and nonstandard form of array assignment:

```
        CALL SYNC_ALL()
        U(1:NN,MM+1) = U(1:NN,1   )[MY_P, MY_QP]  ! North
        U(1:NN,   0) = U(1:NN,  MM)[MY_P, MY_QM]  ! South
        U(NN+1,1:MM) = U(1,   1:MM)[MY_PP,MY_Q ]  ! East
        U(   0,1:MM) = U(  NN,1:MM)[MY_PM,MY_Q ]  ! West
```

Figure 1. Co-Array Halo Assignment between Images

The values `NN` and `MM` are the number of grid points on each node. The values in the square brackets are the coordinates of neighboring images from which data are copied. This copy step keeps the halo current on all nodes. Synchronization is required before the copy step to ensure the data are current.

Mimicking the model of program images resulted in the main thread creating a separate thread for each node. The main thread suspends execution until the images have completed their computation. Global arrays are allocated that contain all halo data. Threads are synchronized *after* the data are stored and then copied using the node's neighboring entries that contain these data:

```
           NORTH_HALO(:,MY_P, MY_Q)=U(1:NN, 1)
           SOUTH_HALO(:,MY_P, MY_Q)=U(1:NN,MM)
            EAST_HALO(:,MY_P, MY_Q)=U(1, 1:MM)
            WEST_HALO(:,MY_P, MY_Q)=U(NN,1:MM)
           CALL SYNC_ALL()

           U(1:NN,MM+1)= NORTH_HALO(:,MY_P, MY_QP)
           U(1:NN, 0)  = SOUTH_HALO(:,MY_P, MY_QM)
           U(NN+1,1:MM) = EAST_HALO(:,MY_PP,MY_Q )
           U( 0,1:MM)   = WEST_HALO(:,MY_PM,MY_Q )
```

Figure 2. Halo Assignment Using Threads and Synchronization

The Co-Array Fortran routine `SYNC_ALL()` is implemented in these tests as a classic barrier function. It uses a counter, a switch, a mutex, and a condition variable. Routines from the Fortran Pthreads package implement this barrier routine.

Computing a residual norm over all images is necessary to establish convergence. When this norm is less than a specific tolerance, in this case $t = 10^{-6}$, then the Jacobi iteration is stopped. The approximate solution is thus sufficiently accurate. Computing the residual norm is done using the following steps:

1. Compute part of the residual contributed by this image.
2. A single image sets the store for the residual norm to zero.
3. Synchronize all images.
4. Each image sets a lock, updates the residual norm, and releases the lock.
5. Synchronize all images.
6. Make the test for convergence using the residual norm.

An equivalent method for computing the residual norm would be to evaluate part of the residual for each image, store this value in a specific global array, synchronize all images, compute the residual norm, again synchronize all images, and then make the test for convergence. This avoids the need to specify a lock for the update step, but requires extra storage for each part of the residual norm. Two synchronization steps are apparently required in either approach.

Results and Discussion

The effectiveness of using Pthreads to solve the Helmholtz equation with Jacobi iteration was examined on the SUN E10000 at the Naval Oceanographic Office MSRC; the IBM Power3, the Origin 2000 (O2K), and the Origin 3000

(O3K) at the ERDC MSRC; and the COMPAQ ES-40 at Aeronautical Systems Center MSRC. The processors/node and the speed of the processors are shown in Table 1.

Table 1. HPC platform characteristics

Platform	Processors/node	Processor speed (MHz)
Origin 3000	512	400
Origin 2000	128	195
SUN E10000	64	400
IBM Power3	8	220
COMPAQ ES-40	4	667

Figures 3-14 show the speedups and efficiencies[1] on the different high performance computing platforms. As can be seen in Figures 3-4, the speedup scales with the number of threads on the SUN E10000 up to approximately 30 threads. Figure 4 shows the thread efficiency on the SUN E10000 to be greater than 90 percent up to approximately 30 threads. For the 8000 by 8000 case the speedup is super-linear up to 16 threads. The SUN E10000 runs were made when most of its 64 processors were in use.

Figure 5 shows the speedup on the IBM Power3 scales reasonably well to approximately eight processors. Since there are eight processors per node on the IBM Power3, scalability up to eight is expected. The corresponding thread efficiency is shown in Figure 6. Beyond six threads, the thread efficiency falls below 90 percent.

The speedups on the O2K scale poorly as can be seen in Figure 7. There are 128 processors per node. This shows the poor thread efficiency as a consequence of the lack of scalability. The reason for the poor scalability on the O2K is not known. Processor contention does not seem to be a reason since the O2K runs were made when more than 100 of the 128 processors were idle. The assumption is that thread priority rules and process stack-size control scalability. These are set to default values at the system level. The effect of stack-size was studied, but no

[1] Define $efficiency = 1 - \left(S_L - S_A\right) / S_L$, where S_L is linear speedup, and S_A is the observed speedup.

improvement was noted by making the size larger. Figures 8-9 show the speedups and efficiency for the O3K. Similar behavior is observed between the O3K and the O2K.

Figures 11-12 show the speedup and efficiency for the COMPAQ ES-40, which has four CPUs/node. A speedup up to four threads is noted, but the efficiency is very poor.

Figure 13 shows the effect of schedule policy on the O2K speedup. Three possibilities were examined, the standard default value (SCHED_OTHER), round-robin (SCHED_RR), and first in first out (SCHED_FIFO). There seems to be no advantage for any of these policies.

Figure 14 shows the wall time comparison between the different HPC platforms. The optimal number of threads is the number of processors per node.

Summary

The specifications of data movement by Co-Array Fortran are replaced by thread programming methods. Each Co-Array image is created from a master program. Communication between the images is achieved in an elementary way by allocating global workspace for the data to be copied. This workspace functions as a buffer area between the images or threads. Thus, to send data from one image or thread to another, use the following copy process:

1. Copy data from image *i* to buffer area, location *i*.
2. Synchronize all images.
3. Copy data from buffer area to data in this image.

In Step 3, data are copied from the buffer space for another thread. The synchronization, Step 2, ensures that each thread's data are current when the copy is completed. This illustrates an easy way to do interthread communication. The weaknesses of this copy-global, copy-local approach includes the following:

- Synchronization may be relatively expensive if implemented with Pthreads. Direct hardware support for synchronizing is preferred, but this is not likely to be portable.
- Each copied data value has its private space in global store. This makes it important to limit the amount of interthread data copy. By increasing the amount of halo data exchanged, it is possible to take more than a single Jacobi iteration before interthread copy. Further limitation on the private halo space that uses locks and signals from Pthreads can be implemented, but this makes for more complex code.

The speedup charts show that thread programming can be effective for this problem. A linear speedup occurs when using the SUN E10000, up to the point where $pq = 49$. For the SGI O2K, the O3K, and the IBM Power3, an approximate linear speedup occurs only for $pq \approx 8$. This is consistent for the IBM Power3 where there are eight CPUs/node. For the O2K and the O3K, the lack of scalability remains a mystery, as there are 128 CPUs/node. Thus, the relative effectiveness of using threads depends heavily on the processor speed, the quality of the underlying thread management firmware, and Pthreads parameters supplied to the runtime system. If one uses the *least elapsed time* as the criterion for the choice of computer system and the number of threads, then the choice becomes a tradeoff between elapsed time and processors required.

These results suggest that a mixed mode of parallel computing strategies may be optimal compared with using message passing or threading alone. In other words, use message passing between programs and within programs use threads for speedups of key code sections. Getting the right mix requires performance evaluation.

References

- Co-Array Fortran is defined at the Web URL http://www.Co-Array.org/
- The example provided is given at the Web URL http://www.Co-Array.org/jacobi.htm
- The Jacobi iterative method is discussed at the Web URL http://www.netlib.org/linalg/html_templates/Templates.html
- Hanson, R. J., Breshears, C. P., Gabb, H. A., "A Fortran Interface to POSIX Threads," Submitted to *ACM-Trans. Math. Software*, July 2000.

Acknowledgement

This work was supported in part by a grant of computer time from the Department of Defense High Performance Computing Modernization Program at the ERDC MSRC, Vicksburg, MS. The authors would like to thank Dr. Wayne Mastin, ERDC MSRC Onsite Academic Lead and Acting PET Director for facilitating the work.

Appendix of Figures

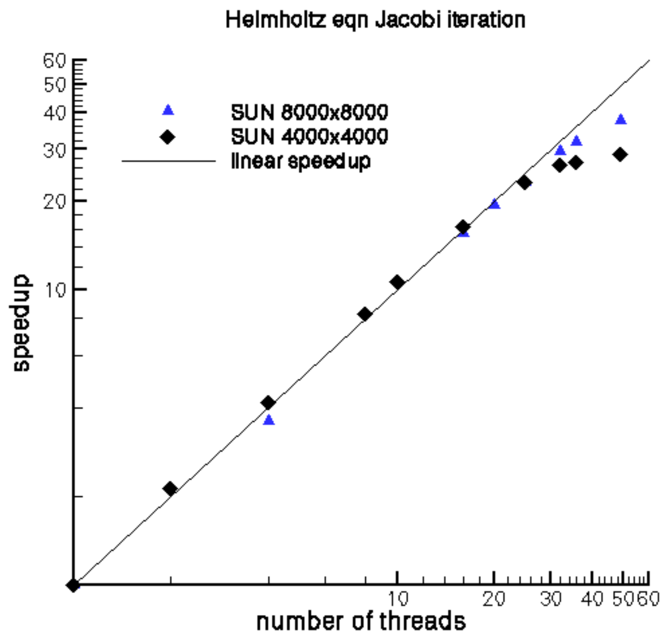

Figure 3. Speedup on the SUN E10000b

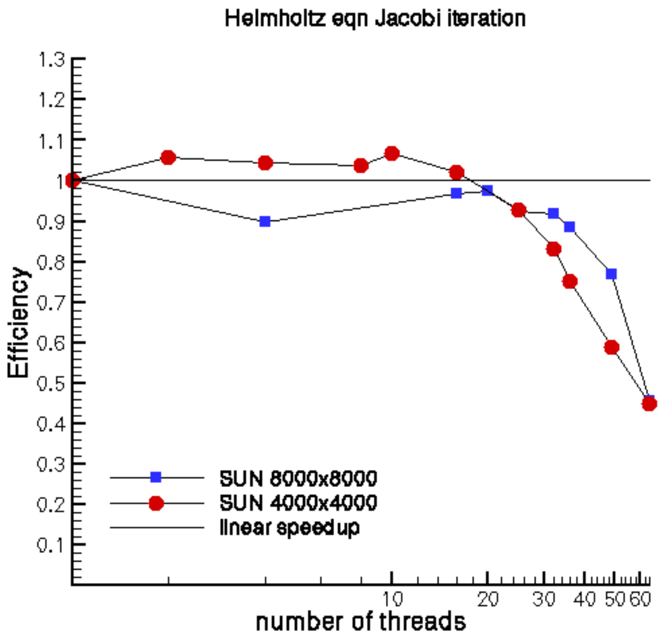

Figure 4. Thread Efficiency on the SUN E10000

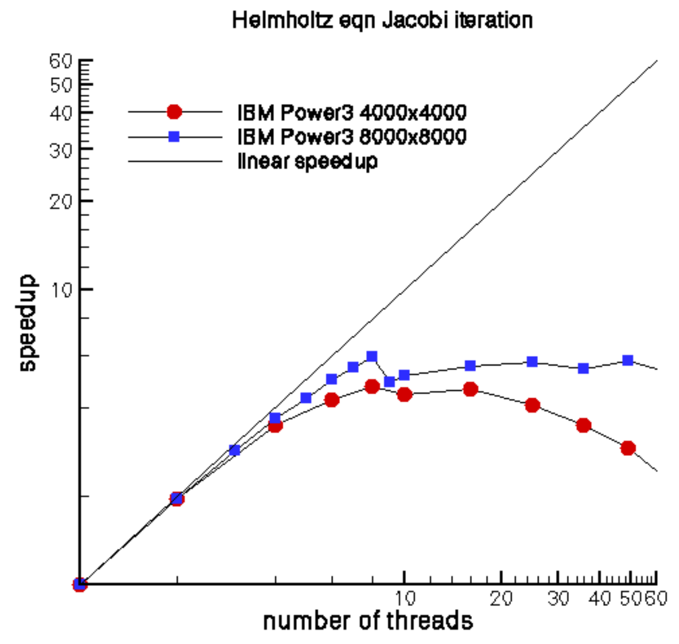

Figure 5. Speedup on the IBM Power3

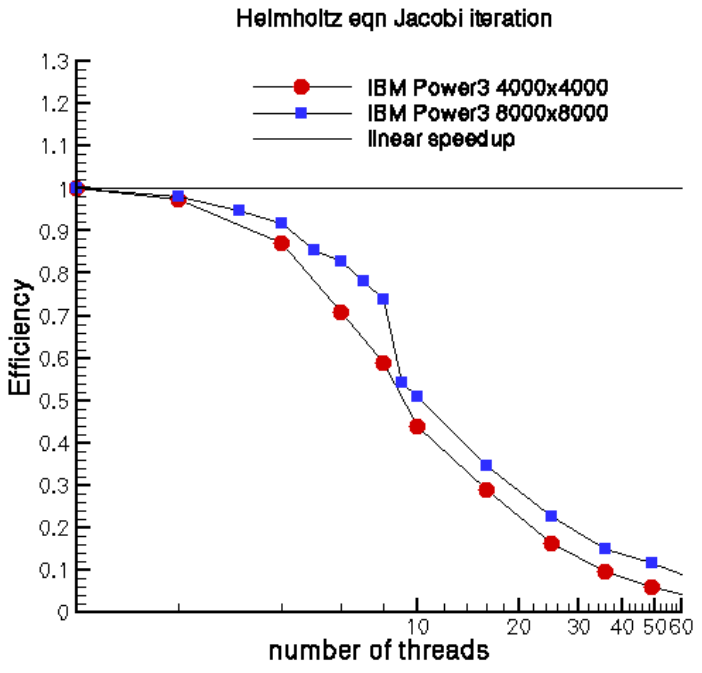

Figure 6. Thread Efficiency on the IBM Power3

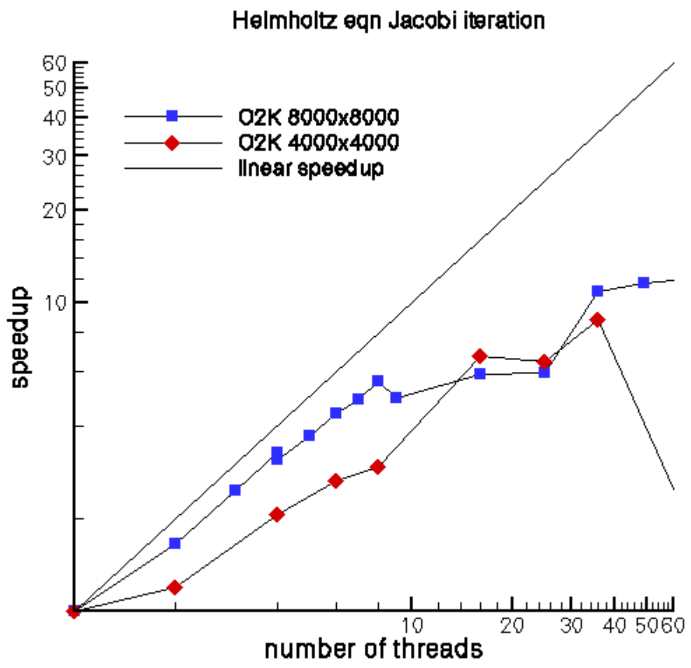

Figure 7. Speedups on the Origin 2000

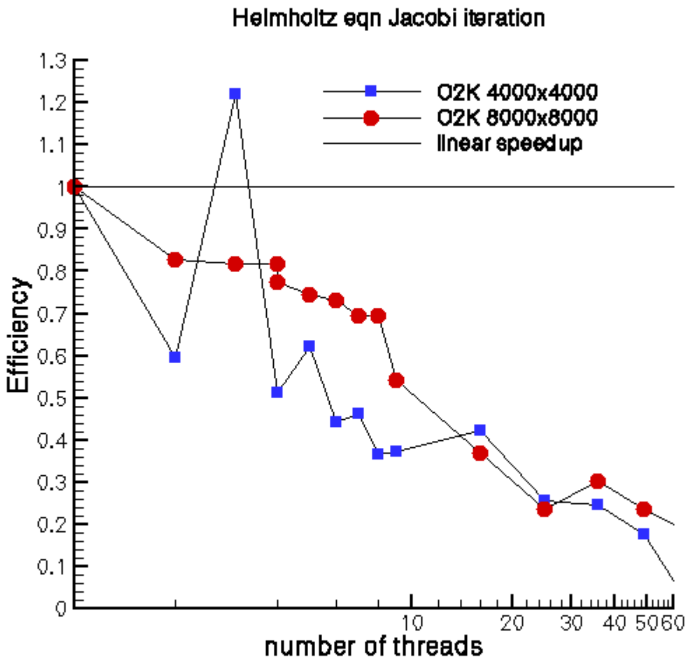

Figure 8. Thread Efficiency on the Origin 2000

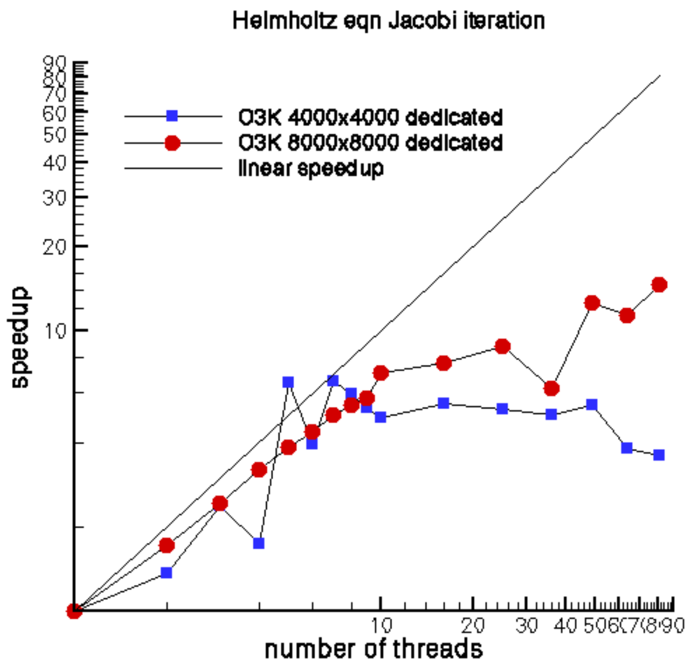

Figure 9. Speedups on the Origin 3000

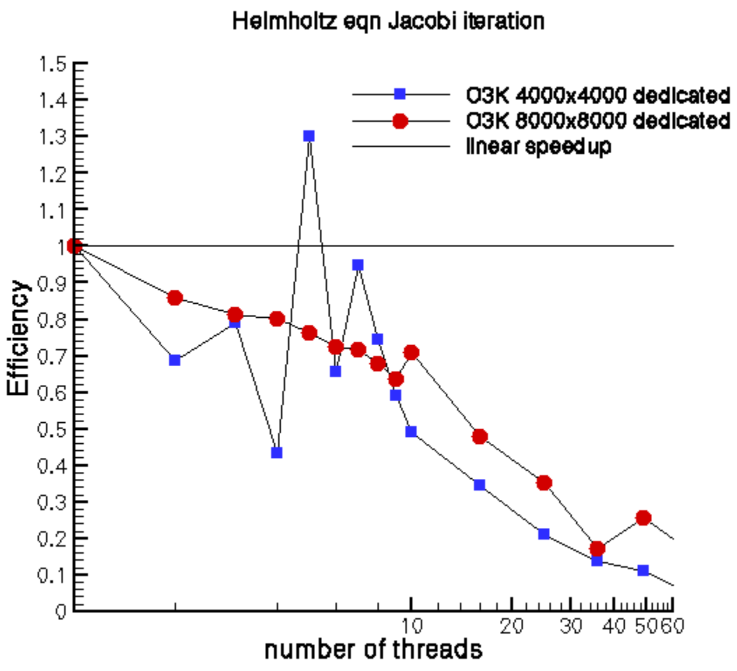

Figure 10. Thread Efficiency on the Origin 3000

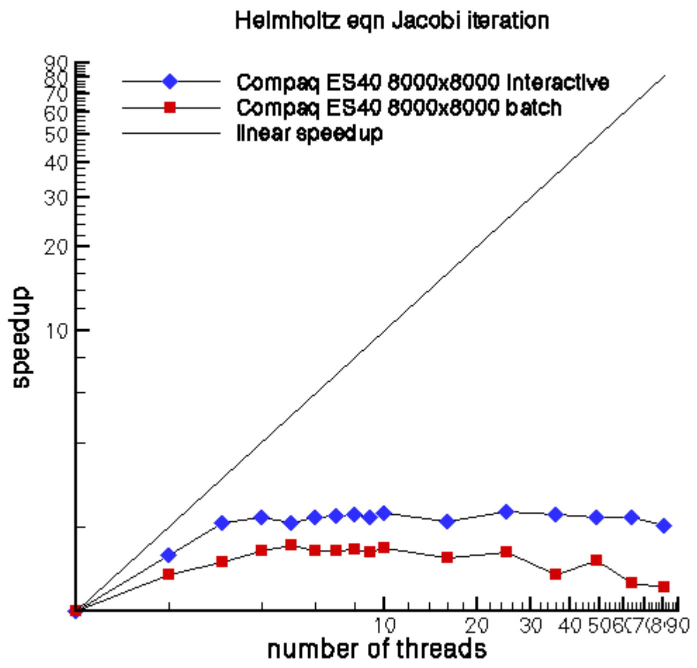

Figure 11. Speedup on the COMPAQ ES 40

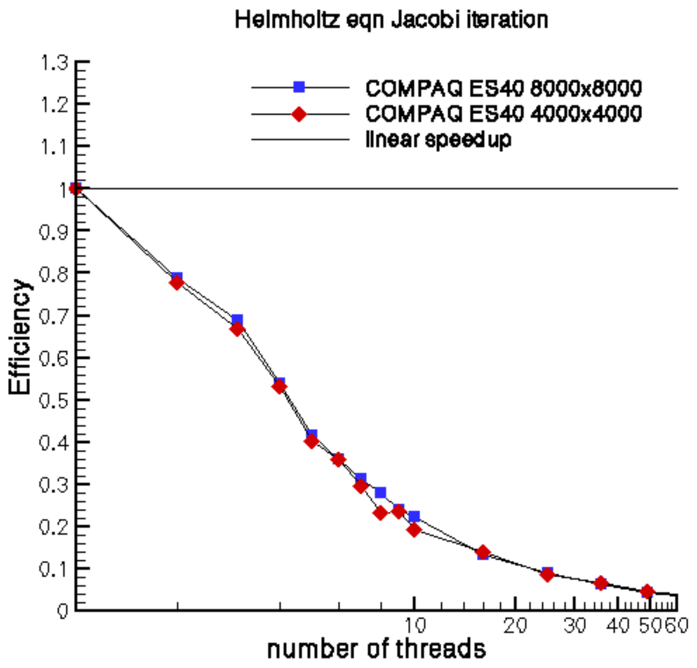

Figure 12. Thread Efficiency on the COMPAQ ES-40

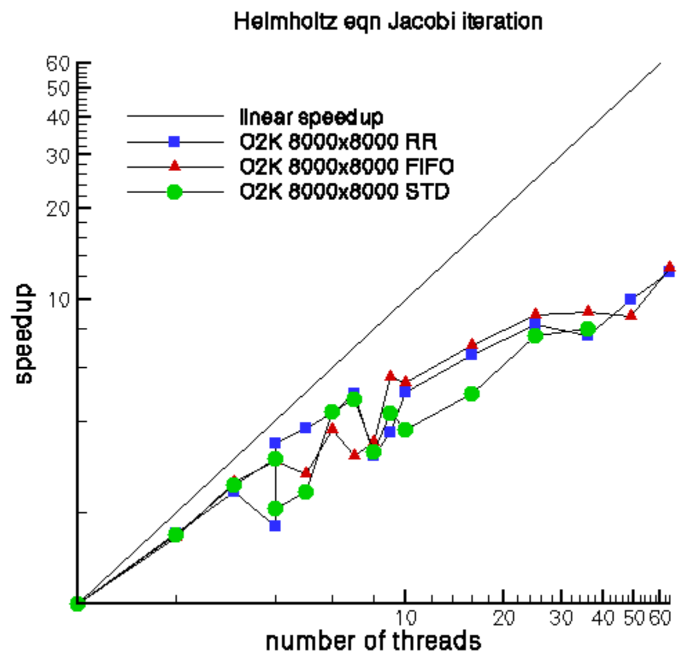

Figure 13. Effect of Schedule Priority on the O2K speedup

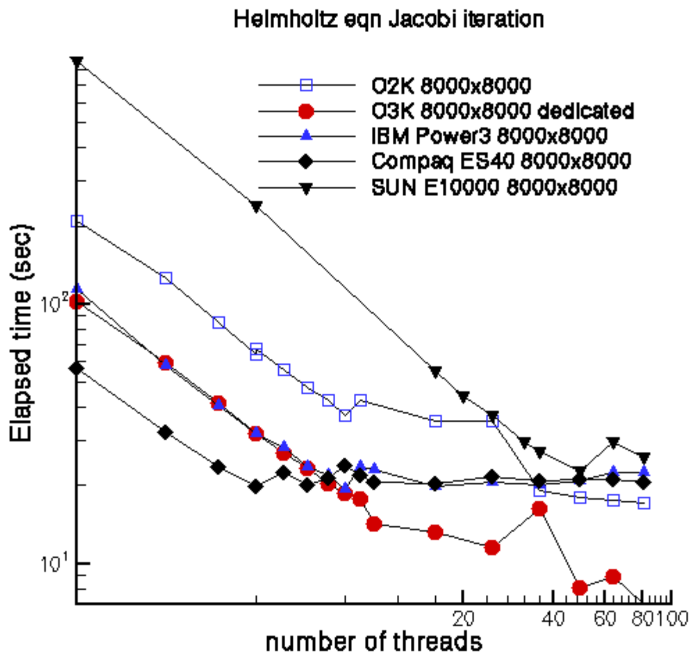

Figure 14. Wall Clock Times on the Different HPC Platforms